Real-Time Performance Issues for linux/linuxRT

Kukhee Kim
ICD Software, SLAC National Accelerator Laboratory

October 23, 2012

Background

- What is linux and linuxRT
  - linux: conventional linux – RHEL5
  - linuxRT: ulibc libarary + RT preemptible kernel patch

- Accelerator Control System has been a territory of the Real-Time Operating System
  - RTEMS, vxWorks, etc

- Now, migrating slowly to linux/linuxRT
  - Linux becomes a player for embedded systems
    - Preemptible kernel, more deterministic behavior (but, not enough yet!)
  - Real-time demand on software is being pushed lower demands on latency
    - Hard real-time part can be taken care of by FPGA
  - Linux has rich supporting tools and applications
  - Human factor
    - Finding good engineer, ….

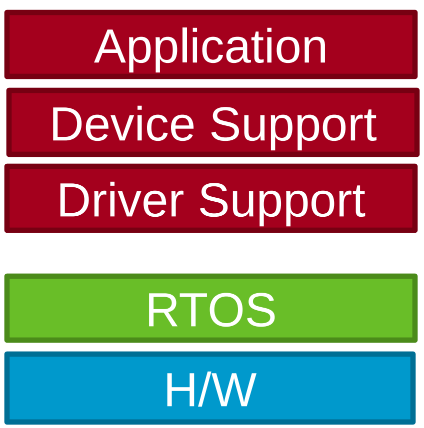
Background Cont'd

- SLAC is just stepping into linux/linuxRT
  - Linux: Camera Applications
  - linuxRT:
    - uTCA Platform: new LLRF and new BPM
    - COM-X Platform: new MCOR

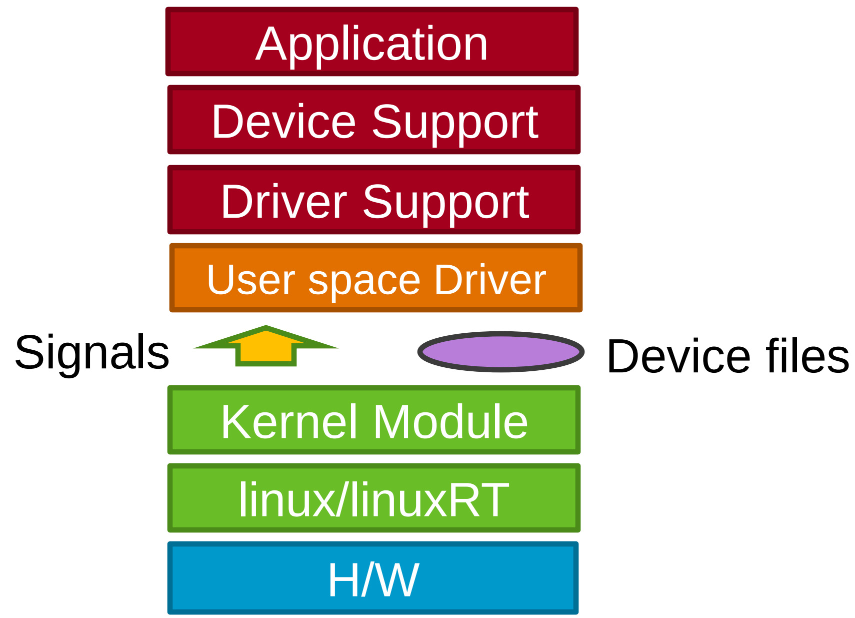
- Faced real-time performance issues, got lot of LESSONs!

Real-Time performance issues in linux/linuxRT

- Kernel space & User space
  - PROS: protect system from user application
  - CONS
    - context switching between kernel level and user level
    - additional steps to access hardware



Software model for RTOS



Software model for linux/linuxRT

Real-time performance issues Cont'd

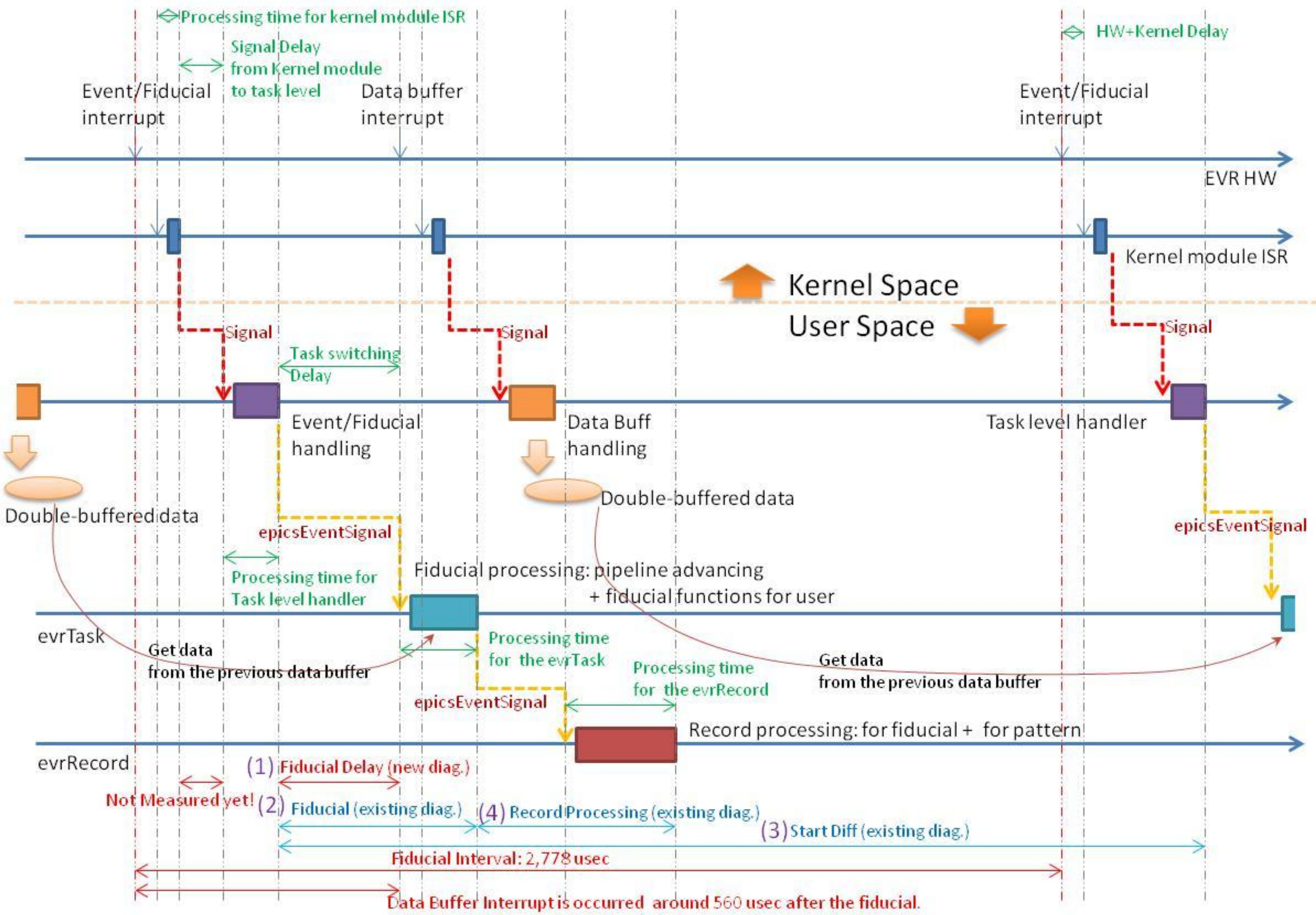
- H/W register, memory access
  - mmap() to user space, can handle it at user space driver
  - no significant difference between RTOS and linux/linuxRT
- Interrupt handling
  - Long processing chain
    - Handler in kernel module -> signal to user space -> user space handler
  - More delay and more jitter
- Signaling from kernel to user space
- RT priority for user space thread and kernel thread

Lessons from the EVR

- MRF Event Receiver (EVR) Hardware + SLAC event module Software
  - Has been used under RTEMS
  - Just ported to linux (PC) and linuxRT (for uTCA)
  - Use MRF kernel module
  - user space driver and a unique software for SLAC (event module)

- SLAC software event module
  - Constrained by complex legacy timing system
    - event pattern
      - 6 x 32 bits modifier, beam code, timeslots, ……
    - pattern pipeline and 360 Hz pipeline advancing
    - Beam Synchronous Acquisition
    - event code and time stamping for each event code
  - Challenging on linux/linuxRT
    - Port existing event software module to this new architecture
  - Interrupt handling & processing is very important
    - >360Hz rate event interrupts + 360Hz rate data buffer interrupts

EVR interrupt handling & processing

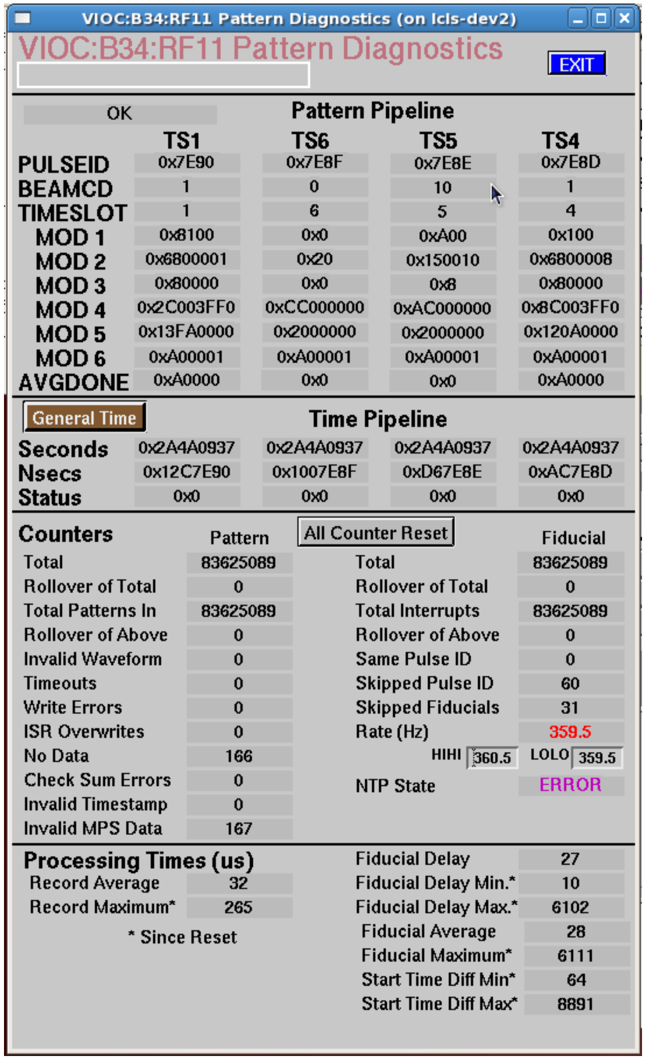


Event interrupt & Data interrupt

- Event interrupt
  - Minimum rate is 360 Hz (because of the fiducial event)
  - More events increase the interrupt rates
  - Fiducial event – fiducial processing
    - Drive the evrTask:
      pattern pipeline advancing + fiducial callback
    - Cascade to the pattern record processing:
      generate EPICS event + pattern diagnostics

- Data interrupt
  - Deliver timing data from EVG to EVR
    - Timestamp, BSA related information, event pattern
    - Used for the next fiducial processing
  - ~350 usec after the fiducial interrupt / 2.43 msec time margin
  - Double buffer handling has been built already to prevent over-writing due to the delayed interrupt or large jitter on the interrupt

Symptoms on linuxRT



(tens hours monitoring)

Fiducial delay : ~ 30 usec (jitter ~ 6,090usec)
Fiducial processing: ~ 30 usec (variation ~ 6,090 usec)
Start Diff: 64usec ~ 8.9 msec (variation ~ 8,830usec)
Record Processing time: ~ 30 usec (max ~ 270 usec)

Messy! Could not meet Real-Time requirements!

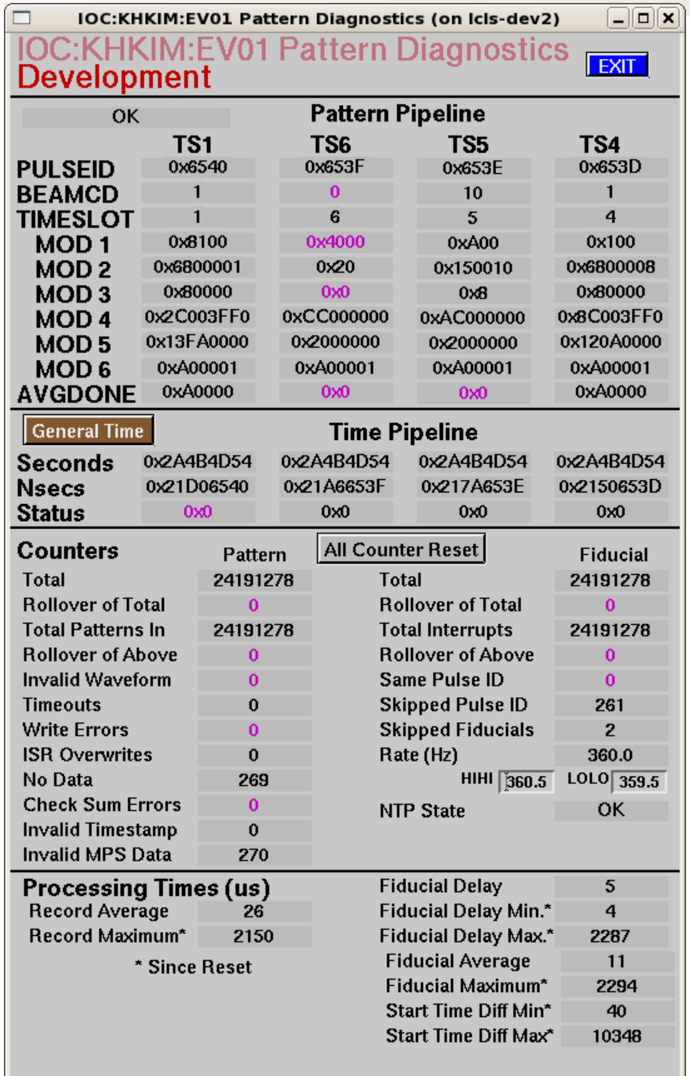
More Analysis

Most of the jitter comes from the epicsEventSignal. It accounts for 6 msec jitter. Please look at the jitter for the fiducial delay.
Another jitter source is the signal between kernel module and the evrTask.
According to the variation of the start diff, we can expect the signal makes ~ 2,740 jitter.
Because,
Var. Start Diff = Jitter. Fiducial Delay + Jitter. Signal Delay
It is almost close to the fiducial interval.

We can just assume, there is no chance (very low possibility) to be preempted during the task level handler and evrTask, due to the following:

(1) Very short processing time < 10 usec
(very low possibility of expiring the scheduling timeslot)
(2) No system call, or scheduler involved :
Actually there is mutex locking but, we are sure it will not be locked at that time.

Symptoms on linux

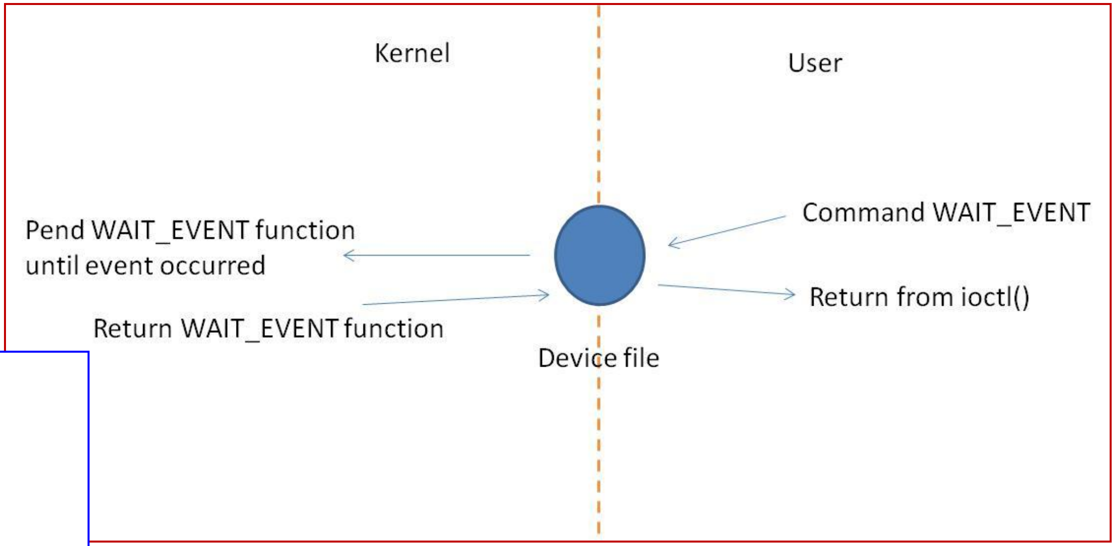


(tens hours monitoring)

Fiducial delay : ~ 10 usec (jitter ~ 2,280usec)
Fiducial processing: ~ 10 usec (variation ~ 2,280 usec)
Start Diff: 40usec ~ 10.4 msec (variation ~ 10,310usec)
Record Processing time: ~ 30 usec (max ~ 2,150 usec)

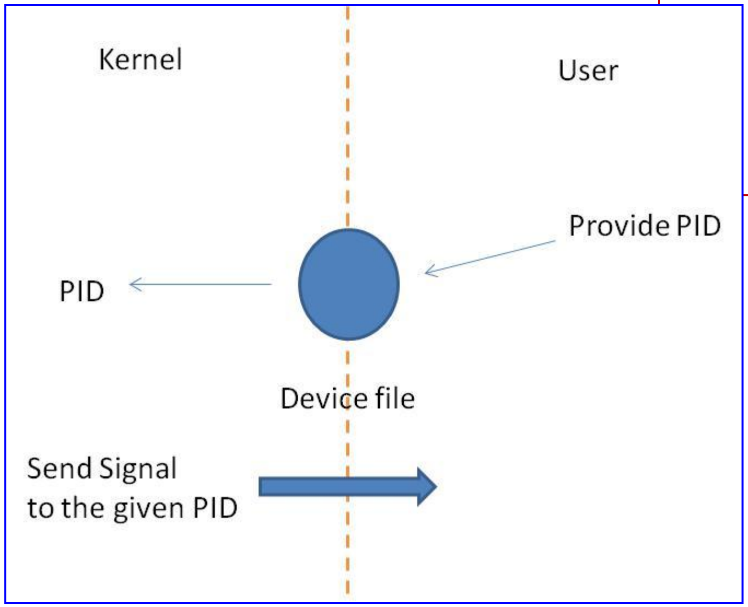
Really Messy!

Signal/Event Propagation between Kernel space & User space



Using ioctl

Using a Signal



EVR kernel module is using a signal instead of the ioctl.

Signal Handling: Asynchronous

- Kernel module generates a signal (SIGIO) to notify an interrupt happens to the User Space Handler

- Original Code
  - The event module registers an signal handler for the SIGIO signal
  - OS makes a software interrupt to switch to the signal handler
  - The SIGIO is handled by the _MAIN_ thread, other threads (epics threads) usually block every signal.
    - _MAIN_ thread: no RT priority + RR Scheduling
    - EPICS thread: RT priority + FIFO Scheduling
  - The signal handler is processed by the _MAIN_ thread without RT priority!
  - Usual case: few tens micro-seconds delay
  - Worst case: few milli-seconds delay!

Signal Handling: Synchronous

- New Code
  - Block the signal from the _MAIN_ thread
  - One epics thread which has high RT priority and waits for the signal in the infinite loop
  - Synchronous: call sigwait() to wait until the signal is received; then scheduling resumes the thread. No software interrupt necessary.
  - Only a few tens of micro-seconds delay

RT Priority for the User threads / *Changes on EPICS

Put the code into the template for <application>Main.cpp

-The hook checks for RT Priority
-If it is unset then, set it to the maximum of SCHED_FIFO

```
#if defined(__linux__) && defined(__GNUC__)

#include <sched.h>
#include <sys/resource.h>

/* We must make sure this code is executed *before*
 * the epicsThread machinery is.
 * Not so easy since the latter eventually is fired
 * up by a c++ static constructor.
 * It is not possible in a portable way to enforce
 * an order of execution among such constructors if
 * they are defined in different compilation units.
 * Under GNU c++ we may use the 'init_priority'
 * attribute.
 * However, if this code is dynamically linked
 * against EPICS base then this doesn't work either
 * since the shared-library is initialized before
 * this code here.
 * Hence we stick a function pointer into the
 * '.preinit_array' section which is executed
 * before any shared libraries are.
 */
static void set_rtprio(int argc, char **argv, char **envp)
{
struct rlimit l;

        if ( getrlimit(RLIMIT_RTPRIO, &l) ) {
                perror("Warning: retrieving real-time priority limits failed");
        }
        /* If the current hard limit is unset the set to the maximum
         * otherwise leave it alone.
         */
        if ( 0 == l.rlim_max )
                l.rlim_max = sched_get_priority_max(SCHED_FIFO);

        l.rlim_cur = l.rlim_max;
        if ( setrlimit(RLIMIT_RTPRIO, &l) ) {
                perror("Warning: setting real-time priority limit failed");
        }
}

static void (*set_rtprio_hook)(int, char**, char**)
__attribute__((section(".preinit_array"),used)) =  set_rtprio;

#endif
```

RT Priority for the Kernel Thread / *Changes on EPICS

- RT preempt patch converts ISR to preemptable kernel thread
  - need to give a proper RT priority for the kernel thread
  - but, the tread will be created after the device open
  - need someway to adjust the RT priority after iocInit()

- Implement the system escape command on the iocsh and execute a shell script to adjust the kernel thread

- New command to run the shell script

```
system("/bin/su root -c `pwd`/rtPrioritySetup.cmd")
```

- Example of the script

```
/usr/bin/chrt -pf 95  `/bin/ps  -Leo pid,tid,rtprio,comm | /usr/bin/awk '/mrfevr/{printf $1}'`
```

RT priority for user threads and kernel threads

SLAC

Adjust RT Priority / Scheduling for Kernel Thread @ linuxRT platform
with `chrt -pf <prio> <pid>`

irqHandler Thread

evrTask

evrRecord

Kernel Thread
to handle the IRQ from EVR

```
$ ps  -Leo pid,ppid,tid,rtprio,stime,time,comm,wchan
  PID  PPID   TID RTPRIO STIME     TIME COMMAND         WCHAN
 1019     1  1019      - 14:58 00:00:00 screen          poll_schedule_timeout
 1020  1019  1020      - 14:58 00:00:00 LLRFControl     n_tty_read
 1020  1019  1022     10 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1024     94 14:58 00:00:39 LLRFControl     sigtimedwait
 1020  1019  1025     10 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1026     69 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1027     58 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1028     63 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1029     70 14:58 00:00:01 LLRFControl     futex_wait_queue_me
 1020  1019  1030     50 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1031     90 14:58 00:00:08 LLRFControl     futex_wait_queue_me
 1020  1019  1032     79 14:58 00:00:09 LLRFControl     futex_wait_queue_me
 1020  1019  1033     59 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1034     69 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1035     59 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1036     60 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1037     61 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1038     62 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1039     63 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1040     64 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1041     65 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1042     16 14:58 00:00:00 LLRFControl     inet_csk_accept
 1020  1019  1043     14 14:58 00:00:00 LLRFControl     hrtimer_nanosleep
 1020  1019  1044     12 14:58 00:00:00 LLRFControl     skb_recv_datagram
 1020  1019  1045     10 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1046     51 14:58 00:00:01 LLRFControl     futex_wait_queue_me
 1020  1019  1047     53 14:58 00:00:00 LLRFControl     skb_recv_datagram
 1020  1019  1049     51 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1052     18 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1053     20 14:58 00:00:00 LLRFControl     sk_wait_data
 1020  1019  1054     18 14:58 00:00:00 LLRFControl     futex_wait_queue_me
 1020  1019  1055     20 14:58 00:00:00 LLRFControl     sk_wait_data
 1023     2  1023     95 14:58 00:00:13 irq/19-mrfevr   irq_thread
```

Misc… / *Changes on EPICS

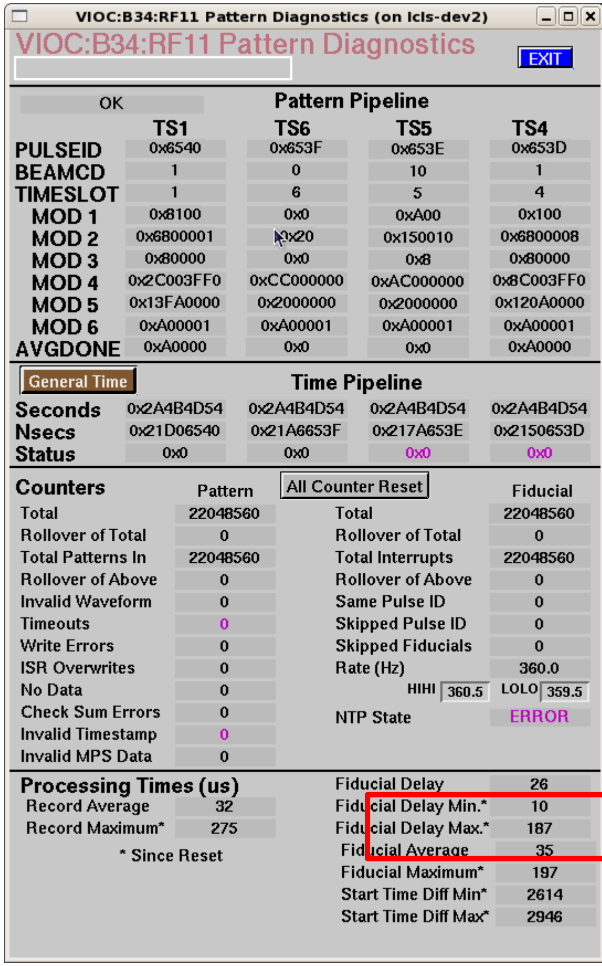
- Memory lock to prevent swapping
  - Put the following code into the main()

```
#if _POSIX_MEMLOCK > 0
   if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1) {
       printf("Fatal error: memory locking fail\n");
       return -1;
   }
#endif
```

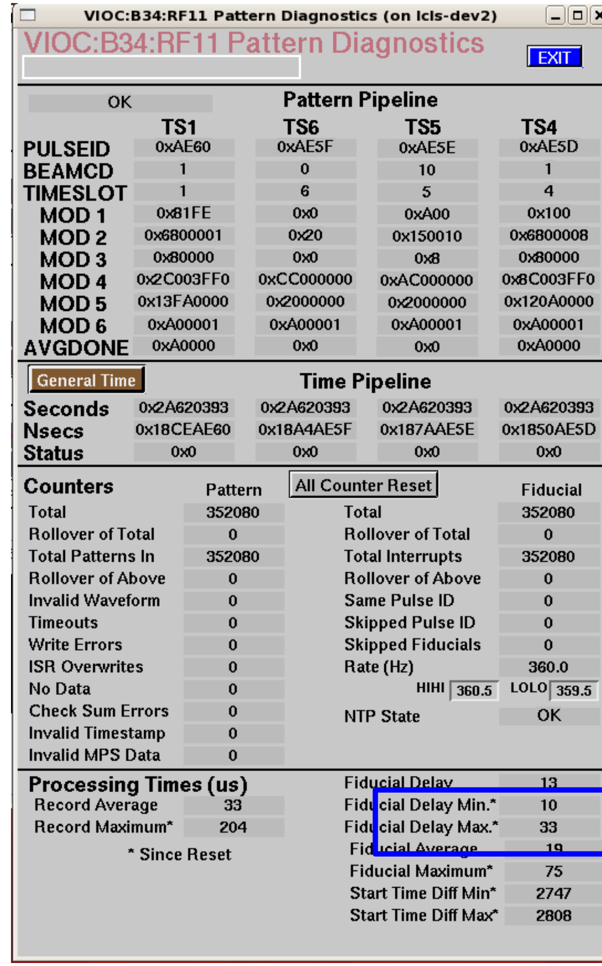
- EPICS POSIX OSI does not support **recursive mutex**
  - The restriction may come from the old version of the Solaris

Result : Significant Improvement on linuxRT

LinuxRT: Switch to synchronous signal handling + adjust the kernel thread priority



before



after

RT Performance Measurement Tool

SLAC

```
epics> dbior drvPerfMeasure 3
Driver: drvPerfMeasure
Estimated Clock Speed: 1500.200532 MHz
Driver has 9 measurement point(s) now...
-------------------------------------------------------------------------------------------
    Node name     Enb          Counter   Time(usec)     Minimum      Maximum          Description
-------------------------------------------------------------------------------------------
        MAINLOOP   1        75120 1970.36458590 1715.28468702 2565.31704789 main thread loop time
             DAQ   1        75158 739.90508357 641.91418378 1056.72872805 get all DAQ data in the main thread
          PHCTRL   1        75157 419.01398286 411.12503752 494.66986857 phase control block in the main thread
      PHCTRLDATA   1        75157 417.50818417 409.73922278 486.73892885   +data get for phase control block in the main thread
          NETDAQ   1        75179   2.83762064   2.55565834  18.90347283         +just net DAQ getting time
          RFDEMO   1        75157 413.68669505 405.80374891 466.58762283         +RF demodulation calc time
      PHCTRLCALC   1        75179   0.46793744   0.30595910  11.60844809   +calculation for the phase control in the main thread
          WFDIAG   1        75175 102.58028625  16.16783855 287.57755433 diag. for waveform in the main thread
         OTHDIAG   1        75179  23.05491783  20.62524265 116.01249052 diag. for others in the main thread
-------------------------------------------------------------------------------------------
```

- Develop RT Performance Measurement Tool
  - measure accurate execution time (nano-second resolution), and worst case latency
  - put probes in the code and enable/disable at runtime
  - very light weight / almost negligible perturbation on the execution
  - now it works as epics driver support
  - device support will be available soon

Summary

- Generic Real-time Performance Issues for linux/linuxRT
  - Kernel space and User Space
  - Interrupt Handling, Signaling to the User Space
  - RT priorities for kernel thread and user thread

- Improvement of EPICS base for linux/linuxRT platform
  - Provide RT priority initialization hook for the user thread (epicsThread)
  - Provide *system escape* to adjust the RT priority for the kernel thread. It can be utilized for others.
  - Memory lock to prevent swapping
  - Recursive Mutex for POSIX OSI in the EPICS

- Develop Real-Time Performance Measurement Tool

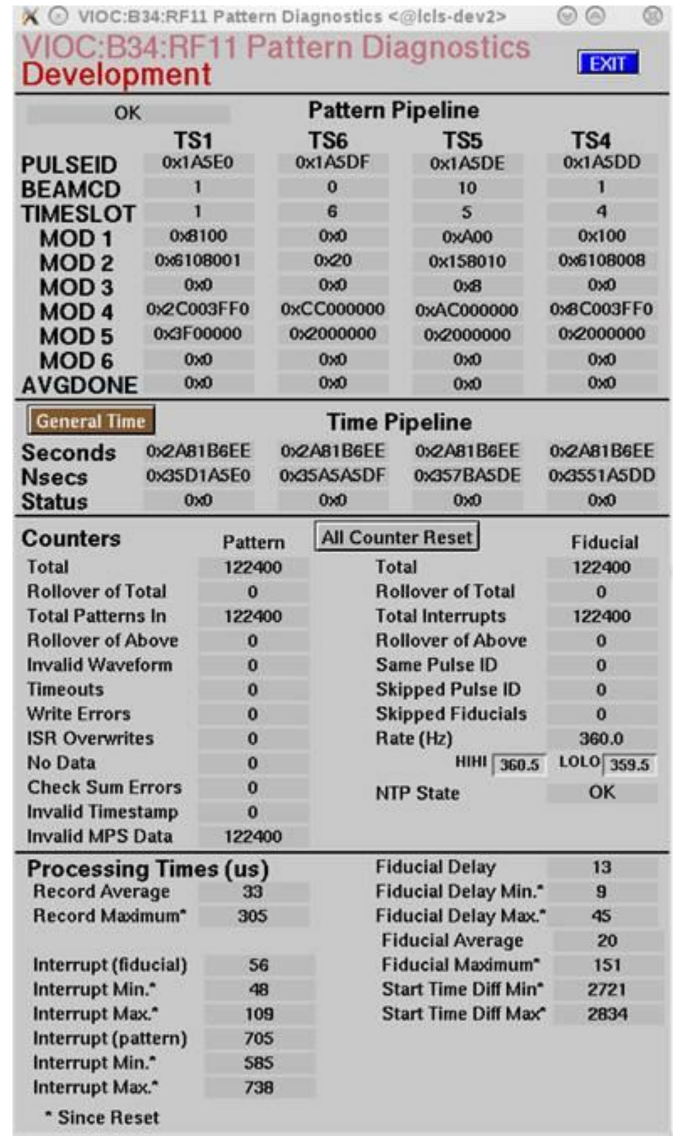
- Acknowledgement
  - Special Thank you to **Till Straumann** and **Stephanie Allison**

Lessons from the work

SLAC

- Put enough DIAGNOSTICS
  to detect, to visualize
- "Pay for Diag."
  It is inexpensive compared to the "Payment for FAIL"

Now, we know the exact delay through H/W -> Kernel -> User ISR :-)



Thank You!